A Relational Dataflow Database.

by

Lubomir Bic

Technical Report #176

Department of Information & Computer Science
University of California, Irvine
Irvine, CA 92717

March 1981

ABSTRACT.

A model of a relational database system based on the principles of functional, data-driven computation is proposed. Relations (sets of data tuples) are represented as streams of values carried by independent tokens among operators of an unraveling dataflow network.

Values may be "updated" by circulating the database through an update operator.
To perform a query on the database, streams involved in that query are replicated and submitted as inputs to dataflow programs (graphs) obtained by translating relational algebra expressions.

All operations, executed by independent processing units, are data-driven and proceed asynchronously as stream elements (needed as inputs) are being produced.
The architecture is composed of a large number of such independent, asynchronously operating processing units, each equiped with a separate memory unit holding a portion of the database.

All processing units are interconnected via 1) a circular shift-register bus, and 2) a daisy chain connection.

Two major techniques are employed to implement the flow of data through the network:
For query processing, the streams required are actually being replicated and physically transported among processing units via the bus. Activity names are used to provide the matching information between streams and operators.
To implement the circulation of the database through the operators described in the model, the operators themselves are moved through the database using the daisy chain connection, thus producing the effect of moving the database.

A time complexity of O(n) is achieved for any relational algebra query assuming a sufficient number of processing units are available.

1. Introduction.

In recent years researchers investigating problems related to functional computing systems have addressed issues related to database applications. The purpose of this paper is to present a database model (and the underlying computer architecture) based on the principles of dataflow /AGP78, Den73/ where

- computation consists of large numbers of asynchronously operating data-driven activities,
- computation is functional (side-effect free).

Our model provides mechanisms to "update" data comprising the database; however, the major emphasis is on the efficiency of information retrieval. We will show how database queries that on conventional single-processor machines execute with $O(n^2)$ time complexity may be carried out in $O(n)$ time, assuming a sufficient number of processing elements are available. In addition, many such queries may be executing simultaneously thus further enhancing the performance of the system.

A database is a conceptual model of the 'real world' organized as a collection of data manipulated via a data-management system. One way to represent information in a database is to organize it in the form of relations (Relational Database Model /Codd70/) where each relation is an unordered set of tuples of related items. Figure 1 shows a sample database comprised of two relations: The relation PRESIDENT, consisting of tuples of the form (pres-name, party, home-state), and the relation ELECTIONS-WON, which associates the election years with the corresponding president name.

PRESIDENT

pres-name	party	home-state
Eisenhower	Republican	Texas
Kennedy	Democrat	Mass.
Johnson	Democrat	Texas
Nixon	Republican	Calif.

ELECTIONS-WON

elect-year	pres-name
1952	Eisenhower
1956	Eisenhower
1960	Kennedy
1960	Johnson
1968	Nixon

Figure 1

This paper considers the aspects of information retrieval and updating under the following assumptions:

- The user must be able to extract information from the database by stating only some conditions or predicates about the data to be retrieved, without having to specify the procedures describing how this data is to be found in the database. A retrieval request specified in such a form may involve searching large portions of the database.
- Since the database is a representation of objects and relations that are changing with time, the user must be able to update information stored in the database. For the purpose of this paper we restrict the update operation to only allow modifications of one tuple at a time. Thus, in order to change (insert, delete) n tuples, n update operations are necessary.
- When retrievals and updates execute simultaneously, the integrity of the database must be preserved.

Relational algebra is a mathematically founded data manipulation language defined over relational databases /Ull80/. It is used to specify what information is to be extracted from a database, independent of the internal representation of the data model.

Relational algebra is defined in terms of 5 basic operations from which database queries of arbitrary complexity may be constructed as relational algebra

expressions. These basic operators are: union, set difference, Cartesian product, restriction, and projection.

The primary reason for choosing relational algebra as a 'programming' language in our considerations was its property of being strictly non-procedural; the user can specify what data is to be extracted from the database without knowledge of how the extraction will be performed. For example, the query

"Find all president names whose home-state is Texas"

when applied to the database from Figure 1 can be expressed as the following relational algebra expression

PRESIDENTS[3="Texas"][1]

with the interpretation: Find all tuples from the relation PRESIDENT whose 3rd field, the home-state field, matches the string "Texas". From these tuples return the 1st field - the president's name.

With the relational expression no statement is made about the order in which the processing of the data should take place.

2. The Database Model.

At the conceptual level, the database consists of streams of values where a stream is an unordered sequence of elements. A stream is the representation of a database relation, each element consisting of exactly one relation tuple. The basic structure of the model is shown in Figure 2.

The stream of requests (req) coming from users contains two types of requests:

1. A retrieval request (query) consists of a relational algebra expression specifying the collection of data to be retrieved.
2. An update request consists of a unique identification of the stream element

(relation tuple) to be updated (relation name and a key value) and the new data to replace (portions of) that element.

All requests are sent to an operator called the request processor.* For each request, conceptually, the entire database will also pass through the request processor. In the implementation of the model, the database is of course not physically being moved, rather the instructions circulate through the database (to be described in section 7).

In the case where a request is a query the database will pass through the request processor and be output on the arc labeled "qry". From this arc the database and the request pass through the operator extract streams which produces copies of those streams involved in the query to be processed. The streams extracted (together with the request) are sent to the next operator perform query which produces the final results. The database itself is returned from the extract stream operator back to the request processor which will accept it with the next request.

In the case where a request is an update operation the database and the request (containing all necessary information) will pass through the request processor and be output on the arc labeled "upd". The operator update then performs the update operation on the element specified as the database passes through the update operator which returns it to the input port of the request processor.

---

* The operators shown in Figure 2 are not elementary operators from the base language.

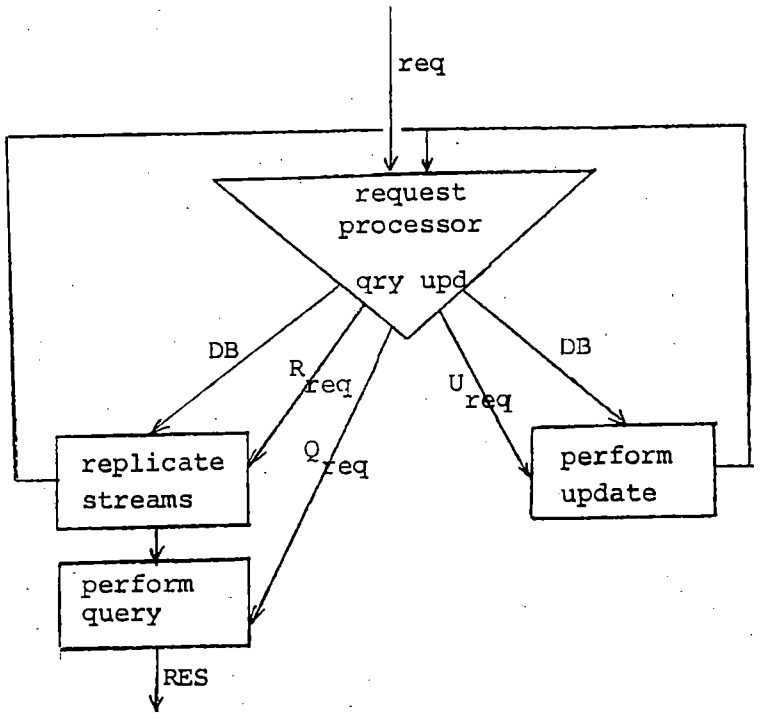


Figure 2

3. Unraveling Interpretation.

The network shown in Figure 2 indicates only the basic interconnections among operators. The actual execution extends into another dimension of parallelism referred to as the unraveling interpretation proposed in /AGP78/:

The streams comprising the database circulate in an infinite loop returning to the request-processor via the "qry" or "upd" arc depending on the type of request. Consider for example a sequence of update requests. According to Figure 2 all elements of the database had to complete their pass through the update operator before the processing of the subsequent update could start. This is an unnecessary constraint. Every operation should be able to start as soon as the first stream element is output by the preceding operation. This principle is depicted in Figure 3 where each of the requests is sent conceptually to a distinct request-processor in the order of the request

arrivals. The database passes through the update or replicate-streams operators and procedes to the next logical request-processor. Thus the network forms an infinite "helix" structure permitting many instances of each operator to operate on (different parts of) the database simultaneously.

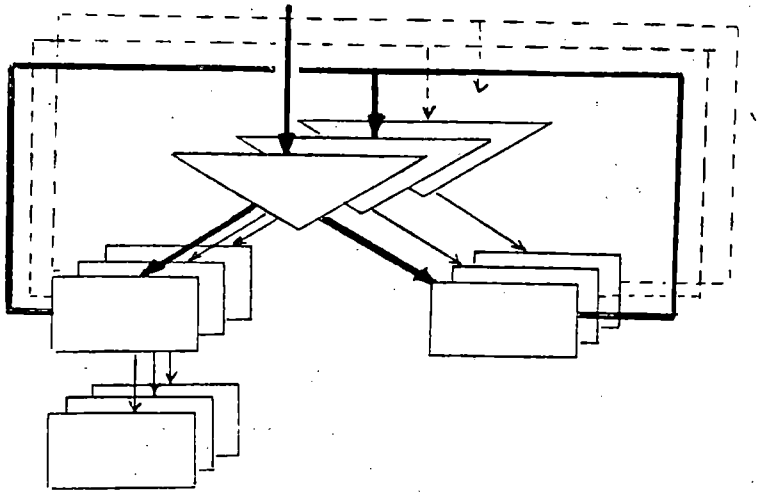


Figure 3

In the following sections 4 and 5 we consider the request-processor and the execution of queries in greater detail.

4. The Request Processor.

The function of the request-processor is to accept requests from users and translate these into elementary programs that can be executed by independent processing units. It operates as follows:

For each retrieval request req (which is a relational algebra expression) it

1. determines which streams are involved in the expression. This information is transmitted to the replicate-stream operator as the stream Rreq where each element of Rreq specifies one stream to be replicated,

2. produces the base language program (graph) and transmits it to the perform-query operator as the stream Qreq where each element of Qreq consists of one elementary operator defined in section 6.

For each update request the request-processor produces a primitive program containing the type of operation (modify, delete, insert), the identification of the stream element to be updated (key value and stream name), and the new data. This primitive program is transmitted to the perform update operator as Ureq.

5. Execution of Queries.

This section is concerned with the perform-query operator which applies relational algebra expressions to streams extracted from the database by the replicate streams operator and produces values satisfying the given relational algebra expression.

The basic idea is to translate relational algebra expressions into base language programs represented as directed graphs consisting of elementary operators interconnected by lines capable of transporting values. Each operator represents an activity that can be executed asynchronously by an independent processing unit. Execution of all operators is strictly data-driven: each operator will produce an output value (a stream element) when and only when at least one element of each input stream has arrived at the operator. The output value is then sent to the next operator expecting that value as input. Thus many operators may simultaneously be engaged in the processing of distinct elements of a stream in a pipelined fashion.

We will define 5 elementary operators and show how these can be interconnected into primitive programs corresponding to the relational algebra operations

union, set difference, Cartesian product, restriction, and projection. The number of operators necessary to form a primitive program is not known at translation time, rather it will be determined at execution time depending on the lengths of the streams being processed. Thus the length of a primitive program (and with that the number of processing units required) is 'demand-driven' - the program will 'grow' according to the number of stream elements being processed.

The time complexity of each of the relational algebra operations (except restrict) on a single-processor machine is $O(n^2)$ where n is the stream length (number of tuples in a relation). By engaging n processing units the time complexity will be reduced to O(n) for all relational algebra operations.
The 5 primitive programs (corrsponding to the 5 relational algebra operations) can then be interconnected to process arbitrary relational algebra expressions (database retrievals). Each of these will execute with the same time complexity O(n) assuming a sufficient number of processing units are available.

6. The Five Relational Algebra Operations.

As mentioned above, all elementary operators operate on values of type stream, the only value type defined in our system. A stream is an unordered sequence of elements each carried by a separate token along the arcs of program grahps. A stream is the representation of a database relation, each stream element consisting of exactly one relation tuple. The number of elements comprising the stream is carried by a special token called the end-of-stream token (eos). For example the relation PRESIDENTS from Figure 1 would be represented as the stream <Eisenhower,Republ.,Texas>,<Kennedy,Democrat,Mass.>,<Johnson,Democrat,Texas>, <Nixon,Republ.,Calif.>, <eos/4>.

Notation: The first element of a stream S that arrives at an operator is denoted as first(S). This could be any element of that stream since no order is imposed on elements within a stream. The remaining elements of S are denoted as rest(S).

6.1. Set Difference.

The set difference of streams (relations) S1 and S2, denoted as S1-S2, is the set of elements in S1 but not in S2. The operation is defined over S1 and S2 if and only if S1 and S2 are of the same degree, i.e. the number of fields in the elements of S1 must be equivalent to the number of fields in the elements of S2.

Elementary Set Difference Operator.

The elem-dif operator is depicted in Figure 4. It accepts 2 streams S1 and S2 as inputs and produces 2 streams S1' and S2' as outputs.

S1' is a copy of S1 with all occurances of the first element of S2 removed (i.e. S1'=S1-first(S2))

S2' is the rest of S2 (i.e. S2'=rest(S2))

If S1 or S2 are empty streams (consisting only of the eos-token) then S1' is also an empty stream and no output is produced for S2'.

The execution of the operator is completely data-driven. The first output will be produced when at least one element of each of the streams S1 and S2 have arrived. Subsequent outputs will be produced upon the arrival of elements from either stream.

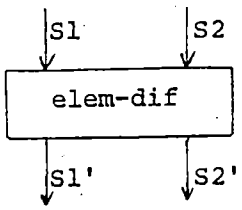


Figure 4

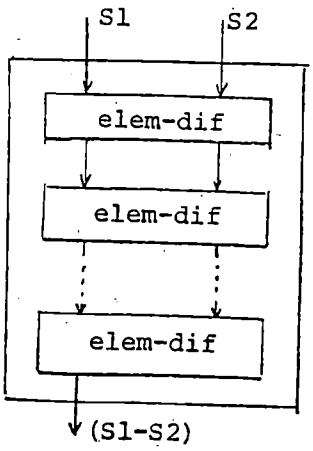


Figure 5

Primitive Program for the Set Difference Operation

The basic operators elem-dif defined above may be combined to form the primitive program for the relational algebra operation set difference as shown in Figure 5.

Each elem-dif operator will remove from S1 all occurances of one of the elements of S2. Since the stream S2 is shortened by one element each time it is passed from one elem-dif operator to the next, it will take n of these operators to form the primitive program for S1-S2 where n is the length of S2.* Thus the length of the primitive program set-difference depends on its inputs and can be determined only at execution time. This is accomplished by allowing primitive programs to "grow" as intermediate streams are being produced by the operators:** As long as intermediate streams are being produced, new operators

---

* In case the set difference S1-S2 results in an empty set before even reaching the end of the stream S2 then the processing can terminate. In that case less than n operators will be required.

** This is similar to the "unraveling" of loops in Id /AGP78/.

are created to process these streams. This process terminates when S1 or S2 results in an empty stream. (Implementation details are discussed in section 7.)

Please note that results produced by each elementary operator are non-deterministic - the resulting streams produced depend on the order of arrival of stream elements at the operator. The computation of the primitive programs (composed of elementary operators), however, is deterministic - the same output streams will be produced for given input streams regardless of the order of arrival of individual stream elements.

6.2. Union.

The union of streams (relations) S1 and S2, denoted as S1 U S2, is the set of elements that occur in S1 or S2 or both. As with the set difference operator, both streams must be of the same degree (same number of fields in each tuple).

Elementary union operator.

The elem-union operator is shown in Figure 6. Similar to the operator elem-dif the operator elem-union accepts 2 streams S1 and S2 as input and produces 2 streams S1' and S2' as output.

S1' is the union of the stream S1 with the first element of S2 that arrives at the operator. In addition, all duplicates of that element are removed from S1 (i.e. S1'=(S1-first(S2)) U first(S2))

S2' is the rest of S2 (i.e. S2'=rest(S2))

If S1 or S2 are empty streams then S1' is empty and no output is produced for S2'.

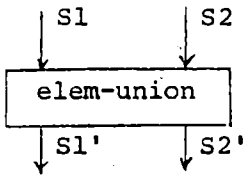


Figure 6

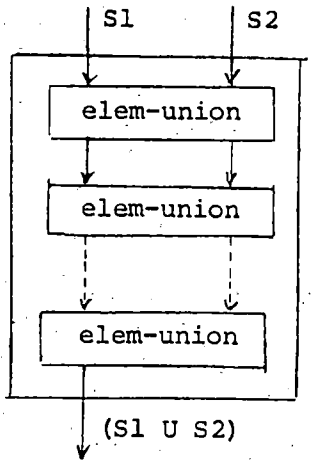


Figure 7

Primitive Program for Union.

By combining n elem-union operators the primitive program for the relational algebra operation union is obtained as shown in Figure 7.

As with the operation set difference the above program will expand according to the length of the stream S2.

Please note that the resulting union S1 U S2 will not contain any duplicate values even if duplicates occured in either of the streams S1 or S2.

6.3. Cartesian Product.

The Cartesian product of streams (relations) S1 and S2, denoted as S1 X S2, is obtained by concatenating each element of S1 with each element of S2.

Elementary Cartesian Product Operator.

The elem-prod operator is shown in Figure 8. It accepts 2 streams S1 and S2 as input and produces 3 streams S1', S1", and S2' as outputs.

S1' is the Cartesian product of the stream S1 and the first element of S2, (i.e. S1'= S1 X first(S2)). No end-of-stream token is produced for the stream S1'.

S1" is an exact copy of S1. It is needed when the primitive program for the Cartesian product is formed. In this case each of the elem-prod operators must receive the complete stream S1 and one element of S2. We chose to propagate the stream S1 through the operators rather than to make separate copies of S1 and distribute them to each elem-prod operator, especially since the number of copies needed can be determined only at execution time.

The stream S2' is the rest of S2 (i.e. S2'=rest(S2))

If S1 or S2 are empty streams then S1' is the eos-token for the complete Cartesian product carrying the count n*m, where n and m are the lengths of S1 and S2 respectively. No output is produced for S1" and S2'.

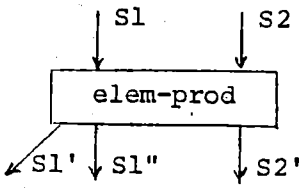


Figure 8

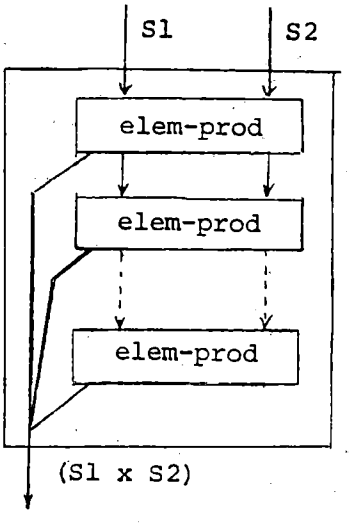


Figure 9

Primitive Program for the Cartesian Product.

Using the elem-prod operators the primitive program for the Cartesian product can be constructed as shown in Figure 9.

Each of the elem-prod operators produces a stream of length n where n is the length of S1. m such streams (all without an eos-token) will be produced, where m is the length of S2. Thus a total of n*m elements is produced and merged into one stream - the Cartesian product S1 X S2. Since the order of elements within a stream is irrelevant, the joining of m lines into one does not require any processing and hence no special 'merge operator' is required. Furthermore, the eos-token for the final stream S1 X S2 is produced by the last elem-prod operator having received an empty stream S2. The final count n*m carried by that token can be calculated by the operator from the length (n) of S1 and the number (m) of elem-prod operators comprising the primitive program. (This number can be deduced from the activity names discussed in section 7.)

6.4. Operators with Parameters.

The last two elementary operators restrict and project comprising the base language are shown in Figure 10 and Figure 11. P and I can be considered paramenters to these operators specifying a particular instance of each of the operators respectively. Thus each of the operators actually represents a class of operators depending on the parameters P or I. (These parameters are known at translation time and become part of the program for the corresponding operator.)

Restriction.

Every stream S consists of elements (tuples) s each of which consists of fields. The restrict operation's function is to eliminate elements of S that do not safisfy a given predicate P applied to individual fields of these elements. This is denoted as S[P].

P is of the form (x op y) where the operands x and y are either field numbers

within elements of S or constant values, and op is one of the comparison operators <, =, >, =/=, =<, >=.

Example: The following restrict operation

PRESIDENT[2="Republican"]

would produce a new relation consisting of those elements of the relation PRESIDENT who's 2nd field matches the constant value "Republican":

<Eisenhower, Republican, Texas>
<Nixon, Republican, Calif.>

The elementary operator restrict which performs the above restrict operation is shown in Figure 10. It accepts a stream S1 as input. Each element of S1 is examined with respect to the predicate P defined above. If the element satisfies P it is output (as part of the new stream S1'), otherwise it is discarded. The last element output is the eos-token reflecting the number of elements in S1'.*

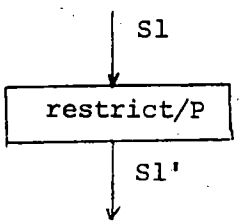


Figure 10

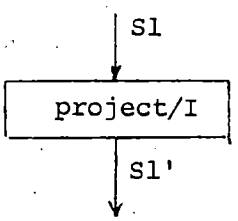


Figure 11

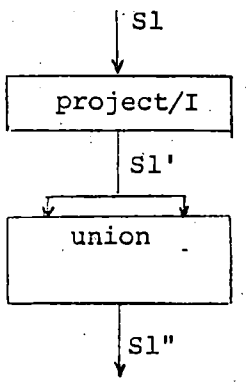


Figure 12

---

* In order to perform the relational algebra operation restrict no composition of operators is necessary - one restrict operator performs the enitre operation. Thus the operator itself can be viewed as the primitive program for the operation restrict.

Projection.

The project operation's function is to remove some fields and/or rearrange some of the remaining fields of each element. A projection may be specified by a sequence of integers i1, i2, ..., in where each ij is the field number (within the original element) to be retained. These fields are to be re-assembled into a new element in the order specified by the given sequence of integers.

The project operation is denoted as S[i1, i2, ..., in].

Example: The projection PRESIDENT[3,1] performed on the database in Figure 1 yields the following stream comprising only the fields 3 and 1 of the original stream:

<Texas, Republican>
<Mass., Democrat>
<Texas, Democrat>
<Calif., Republican>

The elementary operator project which performs a projection is shown in Figure 11, where

S1 is the input stream.

I is a sequence of integers i1, i2, ..., in

S1' is the output stream (of the same length as S1). Each element of S1' is obtained from one element of S1 by selecting and rearranging fields according to the given sequence I.

Elimination of Duplicates.

By omitting fields from the input elements duplicates in the output stream may occur. For example the projection PRESIDENT[2] (Figure 1) would produce a stream of one-field elements two of which had the value "Republican" and the two

others "Democrat".

To eliminate any duplicates from a stream the primitive program for union can be employed as shown in Figure 12.

The "fork" between the two operators indicates that the incoming stream is to be duplicated and sent to both branches of the fork. Thus the stream S1' produced by the operator project will be duplicated and sent to both input ports of the primitive program union.

As pointed out in section 6.2, the primitive program union produces the union of the two input streams while simultaneously eliminating any duplicate values. Hence the stream S1" is the union S1' U S1' with all duplicates removed.

6.5 Composition of Programs.

The primitive operators and primitive programs can be combined to form yet larger programs to process relational algebra expressions of arbitrary complexity. We will demonstrate this principle by showing the program for the following query:

"Retrieve all election years in which a Republican from Texas was elected"

expressed in relational algebra as

(((PRESIDENT[2="Republican"][3="Texas"] X ELECTIONS-WON)[2=4])[1] .

The translation of the expression is straight forward as shown in Figure 13, consisting of 5 primitive programs corresponding to the operations restriction, restriction, Cartesian product, restriction, and projection, specified by the expression.*

---

* Cartesian product followed by a restriction is usually referred to as a join operation. Its time complexity on a single-processor machine is $O(n^2)$.

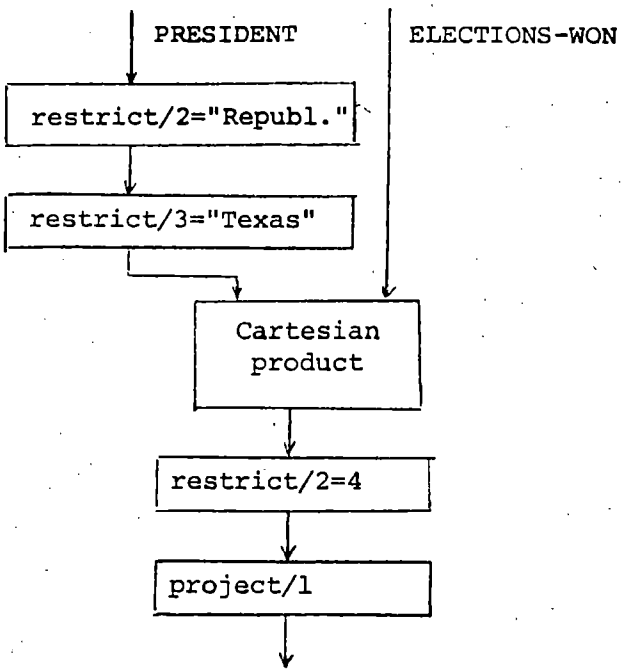


Figure 13

We would like to point out that the time complexity of any relational algebra program such as the above is O(n) (assuming a sufficient number of processing units).

7. Implementation of the Model.

The Architecture.

We consider an architecture similar to that proposed in /AGP78/ as shown in Figure 14.

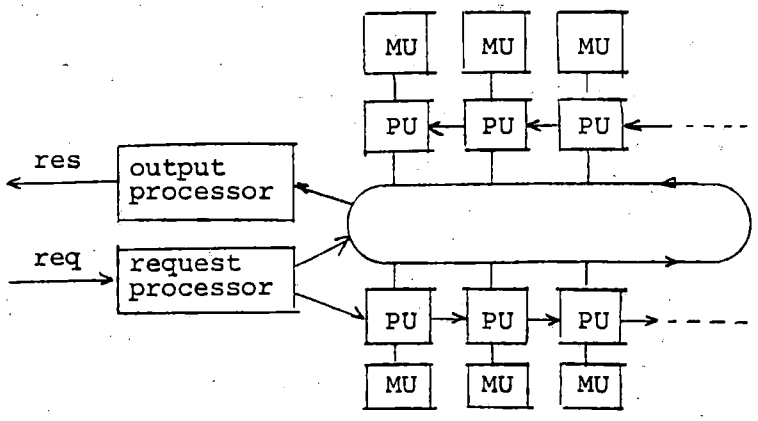


Figure 14

It consists of a large number of processing units (PUs) interconnected via a uni-directional circular shift-register bus. A value put on the bus will eventually pass by all PUs and may be copied or absorbed (removed form the bus) by any of the PUs.

In addition to the bus connection, each PU also has another connection to its neighbor operating in the same direction as the shift-register bus. We will refer to this connection as the daisy chain.

Each PU is equiped with a (large) memory unit MU. The collection of all MUs holds the entire database.* Each of the streams constituting the database should be distributed over as many MUs as possible to enhance parallelism at the physical level.**

The request processor accepts requests and produces the necessary information

---

* In case the size of the database exceeds the physical limits of RAM-technologies fast associative secondary storage devices such as bubble memories may be considered.

** A hashing function on key values is currently under investigation.

packages distributed to PUs. The request processor is the physical implementation of the operator request-processor (Figure 2) described in section 4.

The output processor is responsible for removing streams marked as final results (responses to queries) off the bus and sending them to an output device.

Replication of Streams.

In Figures 2 and 3 the database is shown as circulating for each query through the request-processor and the replicate-streams operator. As the database passes through the replicate-stream operator streams relevant to a query are replicated which is implemented as follows:
For each stream to be replicated the request-processor produces an elementary replicate operator which propagates through all PUs via the daisy chain connection. Thus instead of circulating the database through the replicate-stream operator as shown in the model, the operator itself is circulated through the database producing the same effect.
Upon receipt of the replicate operator each PU will examine its memory unit for any elements belonging to the stream requested. If the memory unit holds any of these elements it will replicate and output these onto the bus.

Operators distributed via the daisy chain may serially follow one another. This will guarantee the unraveling effect of the model (section 3): As soon as an element passes through the replicate-streams operator (which, in the implementation, means that the operator passes through the correspoding PU) it may be used by a subsequent replicate-streams operator (which, in the implementation, means that the next operator may pass through the PU). Thus all streams may be produced simultaneously, delayed only by the propagation time of

the operators along the daisy chain.

(In order to guarantee data integrity no bypassing of operators on the daisy chain is allowed.)

Application of Queries to Replicated Streams.

To perform a query the request-processor will produce a program in the base language composed of primitive programs. Each primitive program consists of n elementary operations where n is known only at execution time.

For illustration purposes consider the relational algebra expression

(S1 U S2)-S3

translated into a base language program shown in Figure 15.

The base language program (graph) must be mapped onto the architecture for execution as follows:

- Each primitive program must be able to "grow" at execution time by reproducing the elementary operator n times according to the input streams as execution progresses. These elementary operators must be mapped onto free PUs for execution.
- Streams must be able to follow the arcs of the graph flowing from one operator to another. In other words, a stream put on the bus by an operator must find its way to the next operator expecting that stream as input.

The following two sections are devoted to the discussion of the above problems.

Mapping Operators onto Free PUs.

Primitive programs for relational algebra expressions consist of n repetitions of an elementary operator (section 6). For each primitive program the corresponding elementary operator will be output on the bus by the

request-processor. (Please note that these operators are not distributed via the daisy chain as was the case with the operators for stream replication. This operator will travel along the bus in the same way as the stream(s) required as input to that operator. A free PU will be allocated to the operator as follows: One element of the input stream is designated as the allocation element. This element will be absorbed by the first free PU it passes by. This PU will then monitor the bus until the corresponding operator arrives. In the same way it will accept the rest of the stream elements needed as input. (In the case where two streams are required as input only one will contain an allocation element.) The stream produced by each elementary operator will allocate the next free PU which in turn will acquire a copy of the same elementary operator as it passes by. This process will repeat until one of the streams becomes empty. At that point the execution of the primitive program is completed and the resulting stream is the input to a subsequent primitive program.

In the above example illustrated in Figure 15, the streams S1, S2, and S3 would be produced by the replicate-stream operation and put on the bus. One of the elements of S1 will serve as the allocation element.
At the same time the two basic operators elem-union and elem-dif will be put on the bus. Depending on the length n of the stream S2 a copy of the elem-union operator will be needed by n distinct PUs. Similarly, m copies of the elem-dif operator will be needed for the subsequent operation. The replication of operators and the allocation of free PUs is done dynamically as follows:
First the allocation element of S1 is absorbed by a free PU. This PU then makes a copy of the corresponding elem-union operator as it passes by. According to the operator the PU will produce the resulting streams, one of which will contain an allocation element. This element will be absorbed by the next free PU which then also makes a copy of the same elem-union operator when it passes

by. This process will repeat until the second stream (propagated as rest(S2)) becomes empty. In this case the elem-union operator is removed from the bus by the PU and discarded. The final stream is the result of the primitive program set-union and becomes the input to the next primitive program set-difference where the same process of duplicating the elem-dif operator will take place until all stream elements have been processed.

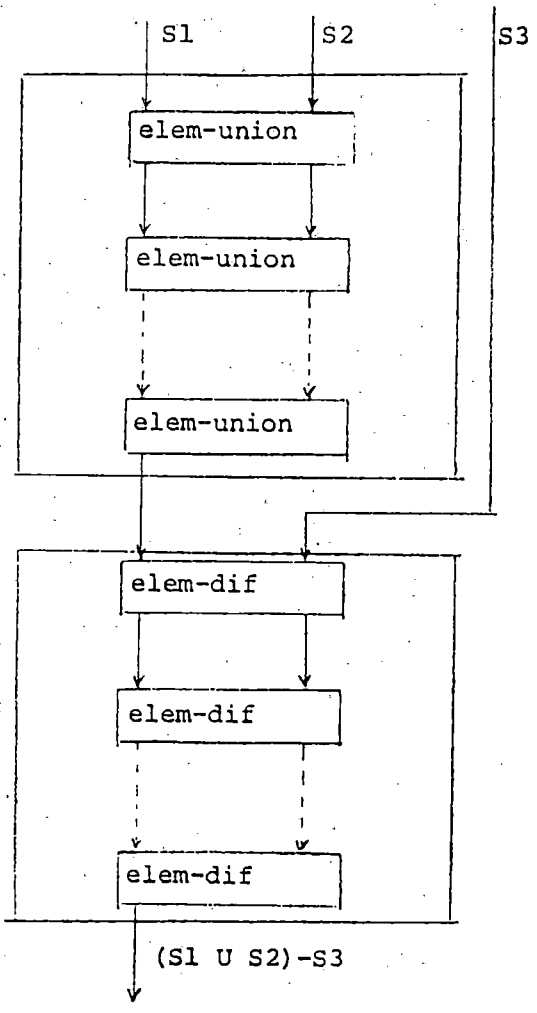


Figure 15

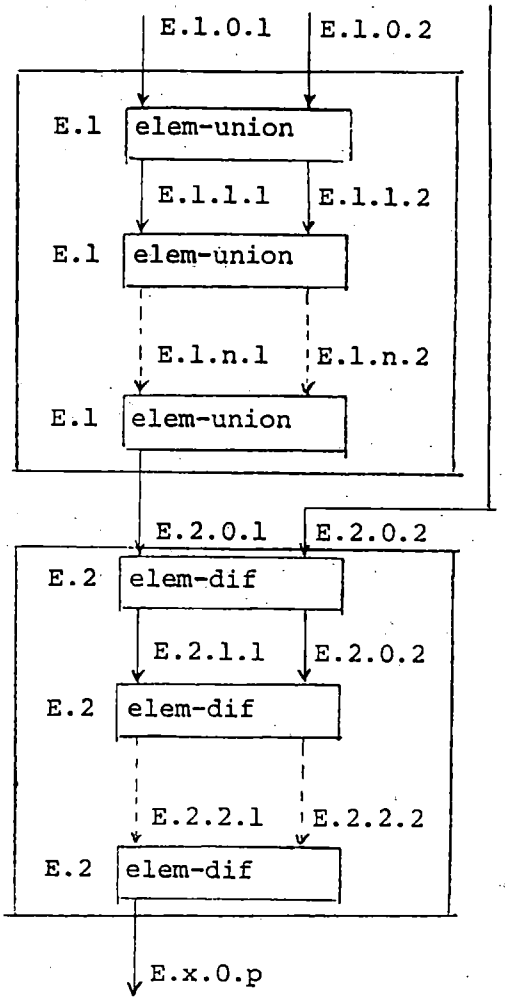


Figure 16

Activity Names.

After a PU has accepted an allocation element it must be able to

- recognize and absorb the rest of the stream,
- determine and copy the corresponding operator,
- determine and absorb the second stream (in the case where 2 streams are required as input).

This is done by means of activity names similar to those described in /AGP78/ which provide the matching information between operators and streams.

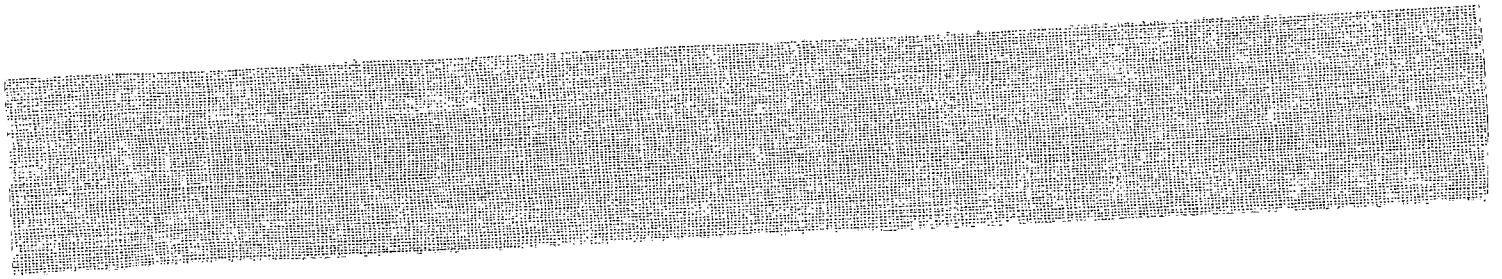
Every stream element carries an activity name of the form E.x.p.i. Every elementary operator such as the above elem-union or elem-dif carries an activity name of the form E.x. The meaning of E, x, p, and i is as follows:

(In the discussion we refer to Figure 16 which shows the manipulation of the activity names for the primitive program from Figure 15.)

E is the number identifying the relational algebra expression being processed. All streams and all operators participating in the computation of the same expression will have the same expression number E. In other words, every base language program has a distinct number E associated with it.

x is called the external number. A distinct external number is assigned to each primitive program by the request-processor at translation time. Thus the elementary operator being replicated and all input streams within the primitive program will have the same external number x. In Figure 16 x equals 1 for all elem-union operators and 2 for all elem-dif operators.

i is called the internal number and is used to control the expansion of primitive programs. The internal number of all stream elements used as inputs to a primitive program will have i=0. Each elementary operator of that primitive program increments i by one as it propagates the streams to the next elementary operator. This process will stop when an elementary operator



receives an empty stream. In this case the activity name of all stream elements produced by that operator will have the form E.x'.p.0 where x' is the external number of the next primitive program expecting these streams as inputs.

Thus the use of activity names provides the implementation of the arcs of a base language graph. At the same time, it provides a mechanism which allows primitive programs to expand according to the streams to be processed.

8. Conclusions.

A database is typically thought of as a large collection of stored data maintained by retrieving and updating its values in an instruction-driven, Von Neuman fashion. This paper is an attempt to present a model for a database represented as a dataflow system, eliminating any memory update operations and providing for a functional data-driven processing of data retrieval requests.

This paper describs only the basic principles of the model and its implementation, and we intend to use it as a basis for further investigations fucussing on actual database applications. Some of the problems involved are:

- data integrity requirements (for individual database operations as well as for transactions composed of several database operations)
- support of different views (subschemas) of a database (including protection and security problems)
- reorganization/restructuring of a database
- architecture/performance issues

References.

/AGP78/ Arvind, K.P.Gostelow, W.Plouffe: "An Asynchronous Programming Language and Computing Machine", Advances in Computing Science and Technology (ed. Ray Yeh), Prentice-Hall publ.

/Codd70/ Codd, E.F.: "A Relational Model of Data for Large Shared Data Banks", Comm. of ACM 13,6 (June 70)

/Den73/ Dennis, J.B.: "First Version of a Dataflow Procedure Language", MAC Tech. Memorandum 61, M.I.T., Cambridge, 1975

/Ull80/ Ullman, J.D.: Database Systems, computer Science Press, 1980